Dahu: Commodity Switches for Direct Connect Data Center Networks

Sivasankar Radhakrishnan*, Malveeka Tewari*, Rishi Kapoor*,
George Porter*, Amin Vahdat†*
* University of California, San Diego † Google Inc.
{sivasankar, malveeka, rkapoor, gmporter, vahdat}@cs.ucsd.edu

ABSTRACT

Solving "Big Data" problems requires bridging massive quantities of compute, memory, and storage, which requires a very high bandwidth network. Recently proposed direct connect networks like HyperX [1] and Flattened Butterfly [20] offer large capacity through paths of varying lengths between servers, and are highly cost effective for common data center workloads. However data center deployments are constrained to multi-rooted tree topologies like Fat-tree [2] and VL2 [16] due to shortest path routing and the limitations of commodity data center switch silicon.

In this work we present Dahu[1], simple enhancements to commodity Ethernet switches to support direct connect networks in data centers. Dahu avoids congestion hot-spots by dynamically spreading traffic uniformly across links, and forwarding traffic over non-minimal paths where possible. By performing load balancing primarily using local information, Dahu can act more quickly than centralized approaches, and responds to failure gracefully. Our evaluation shows that Dahu delivers up to 500% improvement in throughput over ECMP in large scale HyperX networks with over 130,000 servers, and up to 50% higher throughput in an 8,192 server Fat-tree network.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Communications

General Terms

Design; Performance

Keywords

Data center network; Direct connect network

1. INTRODUCTION

Historically, high-speed networks have fallen into two main design spaces. High performance computing (HPC) and supercomputing networks have typically adopted *direct network topologies*, configured so that every switch has some servers connected to it. The remaining ports in each switch are used to connect to other switches in the topology (eg. mesh, torus, hypercube). This type of network is highly resource efficient, and offers high capacity through the presence of many variable-length paths between a source and destination. However, the choice of which path to forward traffic over is ultimately controlled by proprietary protocols in switches, NICs, and by the end-host application logic. This increases the burden on the developer, and creates a tight coupling between applications and the network.

On the other hand, scale-out data centers have adopted *indirect network topologies*, such as folded Clos and Fat-trees, in which servers are restricted to the edges of the network fabric. There are dedicated switches that are not connected to any servers, but simply route traffic within the network fabric. Data centers have a much looser coupling between applications and network topology, placing the burden of path selection on network switches themselves. Given the limited resources and memory available in commodity switches, data center networks have historically relied on relatively simple mechanisms for choosing paths, e.g., Equal-Cost Multi-Path Routing (ECMP).

ECMP relies on static hashing of flows across a fixed set of shortest paths to a destination. For hierarchical topologies like Fat-trees [2], shortest path routing has been largely sufficient when there are no failures. However, recently proposed direct network topologies like HyperX, BCube, and Flattened Butterfly [1, 17, 20], which employ paths of different lengths, have not seen adoption in data centers due to the limitations imposed by commodity data center switches and shortest path routing. ECMP leaves lot of network capacity untapped when there is localized congestion or hot-spots as it ignores uncongested longer paths while forwarding. Further, even in hierarchical networks, ECMP makes it hard to route efficiently under failures, when the network is no longer completely symmetric, and some non-shortest paths can be utilized to improve network utilization.

Commodity switches and shortest path routing have led to hierarchical networks in data centers. These restrictions on topology and routing also mean that higher level adaptive protocols like MPTCP [26] are unable to take advantage of the full capacity of direct networks because all paths are not exposed to them through routing/forwarding tables.

The goal of this paper is to bridge the benefits of direct connect networks—higher capacity with fewer switches (lower cost) for common communication patterns—with the lower complexity, commoditization, and decoupled application logic of data center networks. To that aim, we present Dahu, a lightweight switch mechanism that enables us to leverage non-shortest paths with loop-free forwarding, while operating locally, with small switch state requirements and minimal additional latency. Dahu seeks to obtain the benefits of non-shortest path routing without coupling the application to the underlying topology. Dahu supports dynamic flow-level hashing across links, resulting in higher network utilization. Dahu addresses the local hash imbalance that occurs with ECMP using only local information in the switches.

[1]Dahu is a legendary creature well known in France with legs of differing lengths.

Dahu makes the following contributions: (1) Novel hardware primitives to efficiently utilize non-minimal paths in different topologies with a modest increase in switch state, while preventing persistent forwarding loops, (2) A *virtual port* abstraction that enables dynamic multipath traffic engineering, (3) A decentralized load balancing algorithm and heuristic, (4) Minimal hardware modifications for easy deployability, and (5) Large scale simulations on networks with over 130K servers to evaluate Dahu's performance. Our evaluation shows that Dahu delivers up to 50% higher throughput relative to ECMP in an 8,192 server Fat-tree network and up to 500% throughput improvement in large HyperX networks with over 130,000 servers. We are encouraged by these results, and believe that they are a concrete step toward our goal of combining the benefits of HPC and data center network topologies.

2. MOTIVATION AND REQUIREMENTS

Fully-provisioned multi-rooted tree topologies are ideal for targeting worst case communication patterns—where all hosts in the network simultaneously try to communicate at access link speeds. However, common communication patterns have only few network hot-spots and overprovisioning the topology for worst-case traffic results in high CAPEX. Oversubscribing the multi-rooted tree topology would reduce CAPEX, but network performance would also suffer in the common case since the oversubscribed layers of the tree have even lower capacity to tolerate hot-spots.

Direct networks provide an interesting point in the design space of network topologies, since they provide good performance for most realistic traffic patterns, at much lower cost than fully-provisioned Clos networks [1, pg.8-9][20, pg.6-8]. There are two defining characteristics of direct networks which distinguish them from tree based topologies. (1) Hosts are embedded throughout the structure of the network. Each switch has some hosts connected to it. (2) There are many network paths between any pair of servers—but they are of varying length. These properties of direct networks allow more flexible use of overall network capacity, with slack bandwidth in one portion of the network available to be leveraged by other congested parts of the network by forwarding traffic along longer less congested paths. In a sense the oversubscription is "spread throughout the network" rather than at specific stages or points in the topology.

Direct networks are very popular in HPC—Titan, the world's fastest supercomputer, uses a 3D torus, a direct connect topology [29]. However, data centers have been largely constrained to multi-rooted trees due to commodity switch silicon and shortest path based routing protocols. Direct networks have significant potential in meeting the bandwidth requirements for data centers. Dahu presents simple enhancements to commodity Ethernet switches (both hardware and software) to support direct connect topologies in data centers.

2.1 Challenges

In order to deploy direct connect networks in data centers, we need to address the following challenges:

(1) **Non-shortest path routing:** Current data center switches and routing protocols only support routing over shortest paths. Direct networks, offer large path diversity between servers, but the paths are of varying lengths, typically with only a small number of shortest paths. Shortest path routing artificially constrains bandwidth utilization during periods of localized congestion, and traffic to a destination could potentially achieve higher throughput, if alternate longer paths are also used. Consider a simple mesh network of four switches, as shown in Figure 1. In (a), the shortest path connecting the sources and destinations is congested, by a factor of 3 with a resulting total bandwidth of 1Gbps. However, by sending some traffic flows over non-shortest path links, as shown in (b), the total throughput can be increased to 3Gbps.

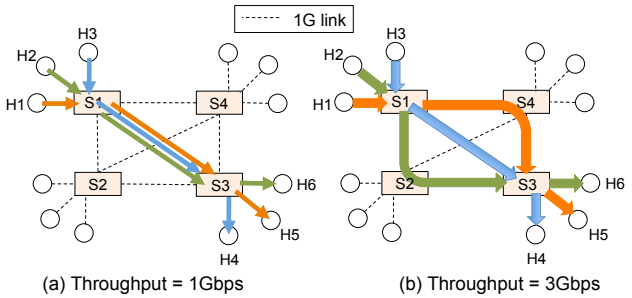


Figure 1: (a) Shortest, and (b) Non-shortest path routing

(2) **Cost-effective commodity switch hardware:** Direct networks in supercomputers rely on custom designed silicon that is expensive and topology dependent. Routing is typically integrated with proprietary protocols in NICs and switches. Data center networks on the other hand are built using low cost commodity off-the-shelf switches [2, 16]. So commodity Ethernet switch silicon must be enhanced to provide the necessary features to support direct connect topologies in the data center while keeping costs low.

(3) **Dynamic traffic management:** Although shortest path routing is the main roadblock to deploying direct networks, the static nature of ECMP style forwarding in switches presents a challenge—even for indirect networks. Hashing flows on to paths, oblivious to current link demands or flow rates can result in imbalance, significantly reducing achieved throughput compared to innate network capacity [3].

There have been several recent proposals for dynamic traffic engineering in data centers. Centralized approaches like Hedera [3] and MicroTE [7] advocate a central fabric scheduler that periodically measures traffic demands, and computes good paths for bandwidth intensive flows to maximize bandwidth utilization. However they have long control loops that only react to changes in traffic at timescales on the order of hundreds of milliseconds at best, i.e. they are only effective for long lived flows. Further, scaling such centralized approaches to very large numbers of flows in large scale data centers presents challenges in terms of switch state requirements. MPTCP [26] is a transport layer solution that splits flows into subflows that take different paths through the network, and modulates the rates of subflows based on congestion. However, in direct networks, MPTCP requires many subflows to probe the different paths, making it impractical for short flows. We illustrate this in Section 6.4.

(4) **Decouple applications and routing:** Direct connect networks in supercomputers tightly couple application development and routing, which requires application developers to be concerned with workloads, routing for certain expected application behaviors, etc. Data center workloads are much more dynamic, and developers often cannot predict the composed behavior of many co-located applications. Handling routing functionality entirely in the network significantly simplifies application development as is done in data centers today.

2.2 Dahu Requirements and Design Decisions

(1) **On-demand non-shortest path routing:** Dahu should only enable non-minimal paths on demand when shortest paths do not have sufficient capacity. Using shorter paths by default results in fewer switch hops and likely lower end-to-end latency for traffic.

Dahu must achieve this while ensuring there are no persistent forwarding loops.

(2) **Dynamic traffic engineering:** Dahu chooses to load balance traffic primarily using local decisions in each switch which helps react quickly to changing traffic demands and temporary hot-spots. In addition, it inter-operates with other routing and traffic engineering schemes.

(3) **Readily deployable:** Any proposed changes to switch hardware should be simple enough to be realizable with current technology, with minimal required switch state. Switches should still make flow-level forwarding decisions—i.e., packets of a particular flow should follow the same path through the network to the extent possible. This avoids excessive packet reordering, which can have undesirable consequences for TCP. Moving flows to alternate paths periodically at coarse time scales (e.g., of several RTTs) is acceptable.

(4) **Generic/Topology Independent:** The switch hardware should be topology independent and deployable in a variety of networks including indirect networks. Non-shortest path routing is also beneficial in the case of Clos topologies which are left asymmetric and imbalanced under failures.

(5) **Fault tolerant:** Failures must be handled gracefully, and rerouting of flows upon failure should only affect a small subset of flows, so that the effect of failures is proportional to the region of the network that has failed. To prevent traffic herds, Dahu should not move many flows in the network around when a single path fails or is congested. Rather, it should be possible to make finer-grained decisions and migrate a smaller subset of flows to alternate paths.

Dahu achieves these targets through a combination of switch hardware and software enhancements, which we describe in Sections 3 and 4 respectively.

3. SWITCH HARDWARE PRIMITIVES

Dahu proposes new hardware primitives which enable better ways of utilizing the path diversity in direct connect topologies, and addresses some of the limitations of ECMP.

3.1 Port Groups With Virtual Ports

ECMP spreads traffic over multiple equal-cost paths to a destination. Internally, the switch must store state to track which set of ports can be used to reach the destination prefix. A common mechanism is storing a list of egress ports in the routing table, represented as a bitmap. Dahu augments this with a layer of indirection: each router prefix points to a set of *virtual ports*, and each virtual port is mapped to a physical port. In fact, the number of virtual ports can be much larger than the number of physical ports. We define a *port group* as a collection of virtual ports mapped to their corresponding physical ports (a many-to-one mapping). The routing table is modified to allow a pointer to a port group for any destination prefix instead of a physical egress port. When multiple egress choices are available for a particular destination prefix, the routing table entry points to a port group.

When the switch receives a packet, it looks up the port group for the destination prefix from the routing table. It computes a hash based on the packet headers, similar to ECMP, and uses this to index into the port group to choose a virtual port. Finally, it forwards the packet on the egress port to which the virtual port is mapped. This port group mechanism adds one level of indirection in the switch output port lookup pipeline, which we use to achieve better load balancing, and support for non-shortest network paths.

In hardware, a port group is simply an array of integers. The integer at index i is the egress port number to which virtual port i in that port group is mapped. Each port group has a fixed number of virtual ports. Multiple destination prefixes in the routing table may point to the same port group. For the rest of this paper, the term *member port* of a port group is used to refer to a physical port which has some virtual port in the port group mapped to it.

The virtual port to egress port mapping provides an abstraction for dynamically migrating traffic from one egress port to another within any port group. Each virtual port is mapped to exactly one egress port at any time, but this mapping can be changed dynamically. A key advantage of the indirection layer is that when a virtual port is remapped, only the flows which hash to that virtual port are migrated to other egress ports. Other flows remain on their existing paths and their packets don't get re-ordered. All flows that map to a virtual port can only be remapped as a group to another egress port. Thus, virtual ports dictate the granularity of traffic engineering, with more virtual ports providing finer grained control over traffic. We propose that each port group have a relatively large number of virtual ports—on the order of 1,000 for high-radix switches with 64-128 physical ports. That means each virtual port is responsible for an average of 0.1% or less of the total traffic to the port group. If required, the routing table can be augmented with more fine grained forwarding entries.

Each port group keeps a set of counters corresponding to each member port indicating how much traffic the port group has forwarded to that port. While having a traffic counter for each virtual port provides fine grained state, it comes at a higher cost for two reasons: (1) The memory required to store the counters is fairly large. For example, a switch with 64 port groups, 1,024 virtual ports per port group, and 64 bit traffic counters needs 512KB of on-chip memory just for these port group counters. (2) Reading all 65,536 counters from hardware to switch software would take a long time, increasing the reaction time of traffic engineering schemes that use all these counters for computations. Dahu uses the port group mechanism and associated counters to implement a novel load balancing scheme described in Section 4.

3.2 Allowed Port Bitmaps

Port groups enable the use of multiple egress port choices for any destination prefix. However, it is sometimes useful to have many egress ports in a port group, but use only a subset of them for forwarding traffic to a particular prefix. One reason to do this is to avoid forwarding on failed egress ports. Consider two prefixes in the routing table F_1 and F_2 both of which point to port group G_1 which has member egress ports P_1, P_2, P_3, P_4. Suppose a link fails and egress port P_4 cannot be used to reach prefix F_1, whereas all member ports can still be used to reach F_2. Now, one option is to create another port group G_2 with member ports P_1, P_2 and P_3 only, for prefix F_1. This can quickly result in the creation of many port groups for a large network which might experience many failures at once. We propose a more scalable approach where we continue to use the existing port group, but restrict the subset of member ports which are used for forwarding traffic to a particular destination.

For each destination prefix in the routing table, we store an *allowed port bitmap*, which indicates the set of egress ports that are allowed to be used to reach the prefix. The bitmap is as wide as the number of egress ports, and only the bits corresponding to the allowed egress ports for the prefix are set. One way to restrict forwarding to the allowed egress ports is to compute a hash for the packet and check if the corresponding port group virtual port maps to an allowed egress port. If not, we compute another hash for the packet and repeat until we find an allowed egress port.

To pick an allowed port efficiently, we propose a parallel scheme where the switch computes 16 different hash functions for the

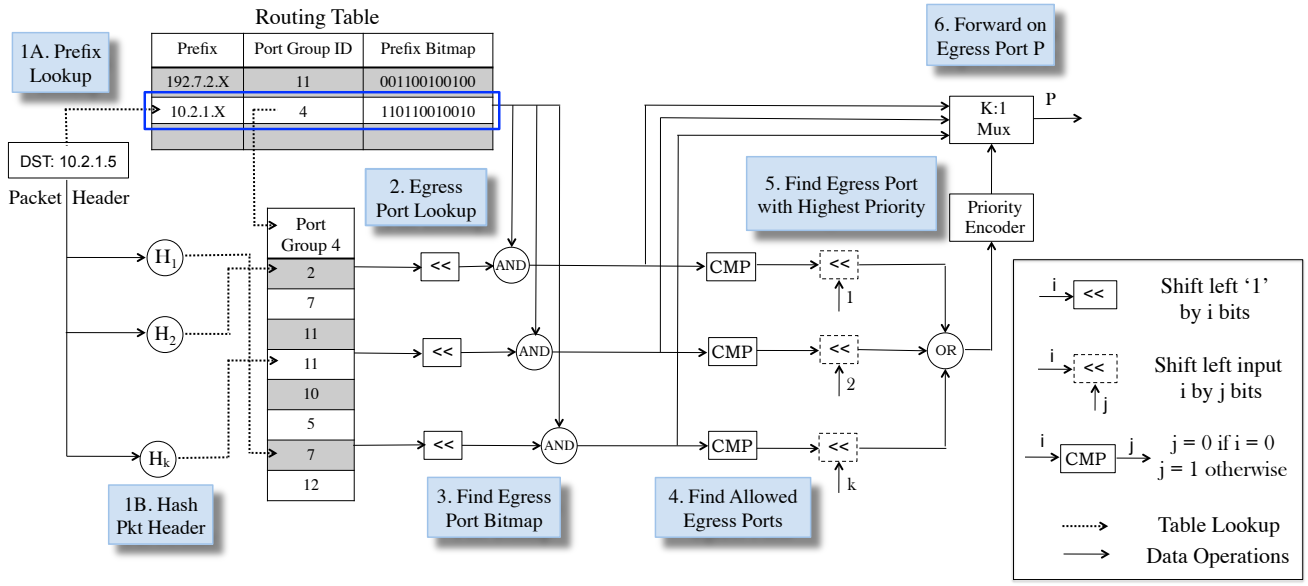


Figure 2: Datapath pipeline for packet forwarding in Dahu.

packet in parallel. The first valid allowed egress port among the hashed choices is used for forwarding. In case none of the 16 hash functions picked an allowed egress port, we generate another set of 16 hash values for the packet and retry. This is repeated some fixed number of times (say 2) to bound output port lookup latency. If an allowed egress port is still not found, we just fall back to randomly picking one of the allowed egress ports, i.e. we ignore the port group mechanism for this packet and just hash it on to one of the allowed egress ports directly. We explore other uses of the allowed port bitmap in Section 4.2.1.

Figure 2 illustrates the egress port lookup pipeline incorporating both port groups and allowed egress port mechanisms. The switch supports a fixed number of allowed port bitmaps for each prefix and has a selector field to indicate which bitmap should be used. Dahu uses an *allow all* bitmap which is a hardwired default bitmap B_{all} where all bits are set, i.e. all member ports of the port group are allowed. The *shortest path* bitmap B_{short} is an always available bitmap that corresponds to the set of shortest path egress ports to reach the particular destination prefix. Unlike the B_{all} bitmap, B_{short} is not in-built and has to be updated by the switch control logic if port group forwarding is used for the prefix. Its use is described in Sections 3.3 and 4.2.1. There can be other bitmaps as well for further restricting the set of egress ports for a destination prefix based on other constraints.

3.3 Eliminating Forwarding Loops

Dahu uses non-shortest path forwarding to avoid congestion hotspots when possible. The term *derouting* is used to refer to a non-minimal forwarding choice by a switch. The number of times a particular packet has been derouted (routed on an egress port not along shortest paths) is referred to as the *derouting count*. An immediate concern with derouting is that it can result in forwarding loops. To prevent persistent forwarding loops, Dahu augments network packets with a 4-bit field in the IP header to store the derouting count. Switches increment this field only if they choose a non-minimal route for the packet. Servers set this field to zero when they transmit traffic. In practice, the derouting count need not be a new header field, e.g., part of the TTL field or an IP option may be used instead.

When a switch receives a packet, if the derouting count in the packet header has reached a maximum threshold, then the switch forwards the packet along shortest paths only. This is enforced using the B_{short} allowed port bitmap for the destination prefix described earlier. The derouting count is also used while computing the packet hash. If a packet loops through the network and revisits a switch, its derouting count will have changed. The resulting change to the hash value will likely forward the packet along a different path to the destination. Each switch also ensures that a packet is not forwarded back on the ingress port that it arrived on. Further, in practice, only a few deroutings are required to achieve benefits from non-minimal routing and the derouting count threshold for the network can be configured by the administrator as appropriate. These factors ensure that any loops that occur due to non-minimal routing are infrequent and don't hinder performance.

As with current distributed routing protocols, transient loops may occur in certain failure scenarios. Dahu uses standard IP TTL defense mechanisms to ensure that packets eventually get dropped if there are loops during routing convergence.

4. SWITCH SOFTWARE

Now we look at how Dahu's hardware primitives can more efficiently utilize the network's available capacity. We describe how to leverage non-minimal paths, and then look at dynamic traffic engineering to address local hash imbalances in switches. These techniques rely on Dahu's hardware primitives, but are independent and may be deployed separately. We begin with some background on HyperX topology.

4.1 Background on HyperX Topology

We use the HyperX topology, a direct connect network for detailing how Dahu's hardware primitives are used, and for evaluating the techniques. This section summarizes the HyperX topology and related terminology [1].

HyperX is an L-dimensional direct network with many paths of varying length between any pair of servers. It can be viewed as a generalization of the HyperCube topology. In an L-dimensional

HyperCube, each switch's position can be denoted by a vector of L coordinates, each coordinate being 0 or 1. Any two switches that differ in their coordinate vectors in exactly one dimension are connected by a bidirectional link. Each switch has some fixed number T of servers attached to it. Eg., a regular cube is a 3-dimensional HyperCube with 8 switches and 12 edges.

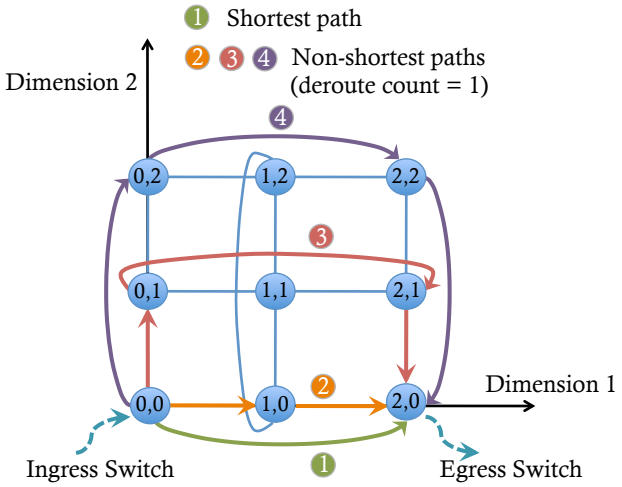


Figure 3: HyperX topology (L=2, S=3). Only switches and the links between them are shown for clarity. The T servers connected to each switch are not shown in the figure. The position of the switches is shown on a 2-dimensional lattice. The paths between switches (0, 0) and (2, 0) with at most 1 derouting are shown. Ingress switch (0, 0) and the egress switch (2, 0) are offset along dimension 1 and aligned along dimension 2.

A regular (L, S, T) HyperX, is a generalization of the HyperCube where the switch coordinates in each dimension are integers in the range [0, S-1] rather than just 0 or 1. Again, any two switches whose coordinate vectors differ in only one dimension are directly connected. Figure 3 shows an example of a 2-dimensional HyperX network with the switches overlaid on a 2-D lattice. An *offset* dimension for a pair of switches is one in which their coordinates differ. Similarly, an *aligned* dimension for a pair of switches is one in which their coordinates are the same. Some examples of HyperX topologies are: (1) A HyperCube is a regular HyperX with S=2, and (2) An L=1 HyperX is just a fully connected graph.

4.2 Non-Minimal Routing

As described earlier, ECMP constrains traffic routes to the set of shortest paths between any pair of switches. While this keeps path lengths low, it can also impose artificial constraints on available bandwidth. Direct connect networks like HyperX have many paths of differing length between any pair of nodes. In a HyperX switch S_s, there are three classes of egress port choices to reach any destination switch S_d.

1. Set of shortest path egress ports to reach S_d. The size of the set is equal to the number of offset dimensions, i.e. dimensions in which the switch coordinates of S_s and S_d differ. In Figure 3, the egress port on switch (0, 0) along path 1 is a short path egress port.
2. Set of egress ports connected to neighbors along offset dimensions excluding the shortest path egress ports. Each of these neighbors is at the same distance from S_d, equal to the shortest path distance from S_s to S_d. In Figure 3, the egress port on switch (0, 0) along path 2 is in this set.
3. All the remaining ports that are connected to other switches. The egress ports which connect to neighbors along dimensions already aligned with S_d are members of this class. Each of these neighbors is one additional hop away from S_d as compared to the shortest path distance from S_s to S_d. In Figure 3, the egress ports on switch (0,0) along paths 3 and 4 are in this set.

Dahu's port group mechanism and allowed port bitmaps enable switches to efficiently route along non-minimal paths. The number of shortest and non-minimal path egress ports for a single destination prefix is not limited artificially, unlike n-way ECMP. We now look in more detail at how to enable non-minimal routing in direct connect networks.

4.2.1 *Space saving techniques*

A strawman solution for non-minimal routing is to create one port group for each destination prefix. For each prefix's port group, we make all appropriate physical ports (both along shortest paths and non-minimal paths to the destination) members of the port group. When a switch receives a packet, it looks up the port group for the destination prefix, and hashes the packet onto one of the virtual ports in the port group. Use of some of the corresponding egress ports results in non-minimal forwarding.

One characteristic of the HyperX topology is that in any source switch, the set of shortest path egress ports is different for each destination switch. These ports must be stored separately for each destination switch, thereby requiring a separate destination prefix and a separate port group in case of the strawman solution. For a large HyperX network, the corresponding switch memory overhead would be impractical. For example, a network with 2,048 128-port switches, and 1,024 virtual ports per port group would need 2 MB of on-chip SRAM just to store the port group mapping (excluding counters). Thus, we seek to aggregate more prefixes to share port groups. We now describe some techniques to use a small number of port groups to enable the use of non-minimal paths, while using only shortest paths whenever possible.

In a HyperX switch, any egress port that is connected to another switch can be used to reach any destination. However not all egress ports would result in paths of equal length. Let us assume that we only use a single port group PG_{all}, say with 1,024 virtual ports. All physical ports in the switch connected to other switches are members of the port group PG_{all}. If we simply used this port group for all prefixes in the routing table, that would enable non-minimal forwarding for all destinations.

Dahu uses the allowed port bitmap hardware primitive to restrict forwarding to shorter paths when possible. If Dahu determines that only shortest paths need to be used for a particular destination prefix, the B_{short} allowed port bitmap for the prefix is used for forwarding, even when the derouting count has not reached the maximum threshold. Otherwise, the allowed port bitmap is expanded to also include egress ports that would result in one extra hop being used and so on for longer paths. For HyperX, there are only three classes of egress port choices by distance to destination as described earlier; in our basic non-minimal routing scheme, we either restrict forwarding to the shortest path ports or allow all paths to the destination (all member ports of PG_{all}).

We now look at the question of how Dahu determines when additional longer paths have to be enabled for a destination prefix to meet traffic demands. Switches already have port counters for the total traffic transmitted by each physical port. Periodically (e.g.,

1. Compute aggregate utilization (Agg) and capacity (Cap) for all egress ports in the B_{short} bitmap
2. If Agg / Cap < Threshold,
   Set allowed ports to B_{short} (there is sufficient capacity along shortest paths for this prefix)
   Else, set allowed ports to B_{all}.

Figure 4: Restricting non-minimal forwarding

every 10ms), the switch software reads all egress port counters, iterates over each destination prefix, and performs the steps shown in Figure 4 to enable non-minimal paths based on current utilization. It is straightforward to extend this technique to progressively enable paths of increasing lengths instead of all non-minimal paths at once. In summary, we have a complete mechanism for forwarding traffic along shorter paths whenever possible, using just a single port group and enabling non-minimal routing whenever required for capacity reasons.

4.2.2 Constrained non-minimal routing

As described earlier, in a HyperX network, each switch has three classes of egress port choices to reach any destination. Based on this, Dahu defines a constrained routing scheme as follows—a switch can forward a packet only to neighbors along offset dimensions. If a packet is allowed to use non-minimal routing at a switch, it can only be derouted along already offset dimensions. Once a dimension is aligned, we do not further deroute the packet along that dimension. After each forwarding choice along the path taken by a packet, it either moves closer to the destination or stays at the same distance from the destination. We call this scheme *Dahu constrained routing*. For this technique, we create one port group for each possible set of dimensions in which the switch is offset from the destination switch. This uses 2^L port groups where L, the number of dimensions is usually small, e.g., 3–5. This allows migrating groups of flows between physical ports at an even smaller granularity than with a single port group.

This technique is largely inspired by Dimensionally Adaptive, Load balanced (DAL) routing [1]. However, there are some key differences. DAL uses per-packet load balancing, whereas Dahu uses flow level hashing to reduce TCP reordering. DAL allows at most one derouting in each offset dimension, but Dahu allows any number of deroutings along offset dimensions until the derouting threshold is reached.

4.3 Traffic Load Balancing

Per-packet uniform distribution of traffic across available paths from a source to destination can theoretically lead to very good network utilization in some symmetric topologies such as Fat-trees. But this is not used in practice due to the effects of packet reordering and faults on the transport protocol. ECMP tries to spread traffic uniformly across shortest length paths at the flow level instead. But due to its static nature, there can be local hash imbalances. Dahu presents a simple load balancing scheme using local information at each switch to spread traffic more uniformly.

Each Dahu switch performs load balancing with the objective of *balancing* or *equalizing* the aggregate load on each egress port. This also balances bandwidth headroom on each egress port, so TCP flow rates can grow. This simplifies our design, and enables us to avoid more complex demand estimation approaches. When multiple egress port choices are available, we can remap virtual ports between physical ports, thus getting fine grained control over traffic. Intuitively, in any port group, the number of virtual ports that map to any member port is a measure of the fraction of traffic from the port group that gets forwarded through that member port. We now describe the constraints and assumptions under which we load balance traffic at each switch in the network.

4.3.1 Design Considerations

Periodically, each switch uses local information to rebalance traffic. This allows the switch to react quickly to changes in traffic demand and rebalance port groups more frequently than a centralized approach or one that requires information from peers. Note that this design decision is not fundamental–certainly virtual port mappings can be updated through other approaches. For different topologies, more advanced schemes may be required to achieve global optimality such as through centralized schemes.

We assume that each physical port might also have some traffic that is not *re-routable*. So Dahu's local load balancing scheme is limited to moving the remainder of traffic within port groups. Dahu's techniques can inter-operate with other traffic engineering approaches. For example, a centralized controller can make globally optimal decisions for placing elephant flows on efficient paths in the network [3], or higher layer adaptive schemes like MPTCP can direct more traffic onto uncongested paths. Dahu's heuristic corrects local hashing inefficiencies and can make quick local decisions within a few milliseconds to avoid temporary congestion. This can be complemented by a centralized or alternate approach that achieves global optimality over longer time scales of few hundreds of milliseconds.

4.3.2 Control Loop Overview

Every Dahu switch periodically rebalances the aggregate traffic on its port groups once each epoch (e.g., every 10ms). At the end of each rebalancing epoch, the switch performs the following 3 step process:

Step 1: **Measure current load**: The switch collects the following local information from hardware counters: $(1a)$ for each port group, the amount of traffic that the port group sends to each of the member ports, and $(1b)$ for each egress port, the aggregate bandwidth used on the port.

Step 2: **Compute balanced allocation**: The switch computes a balanced traffic allocation for port groups, i.e. the amount of traffic each port group should send in a balanced setup to each of its member ports. We describe two ways of computing this in Sections 4.4 and 4.5.

Step 3: **Remap port groups**: The switch then determines which virtual ports in each port group must be remapped to other member ports in order to achieve a balanced traffic allocation, and changes the mapping accordingly. We have the current port group traffic matrix (measured) and the computed balanced traffic allocation matrix for each port group to it member egress ports.

As mentioned in Section 3.1, a switch only maintains counters for the total traffic from a port group to each of its member ports. We treat all virtual ports that map to a particular member port as equals and use port group counters to compute the average traffic that each of the virtual ports is responsible for. Then, we remap an appropriate number of virtual ports to other member ports depending on the intended traffic allocation matrix using a first-fit heuristic. In general, this remapping problem is similar to bin packing.

4.4 Load Balancing Algorithm

We now describe an algorithm for computing a balanced traffic allocation on egress ports. Based on the measured traffic, the

PG \ Port	0	1	2	3
0	4	1	2	--
1	--	1	--	2
BG	2	2	2	0
Agg	6	4	4	2

Initial Port Group (PG) Utilizations

PG \ Port	0	1	2	3
0	2 ⅔	1 ⅔	2 ⅔	--
1	--	1	--	2
BG	2	2	2	0
Agg	4 ⅔	4 ⅔	4 ⅔	2

Step 1: Balancing Port Group 0

PG \ Port	0	1	2	3
0	2 ⅔	1 ⅔	2 ⅔	--
1	--	0	--	3
BG	2	2	2	0
Agg	4 ⅔	3 ⅔	4 ⅔	3

Step 2: Balancing Port Group 1

PG \ Port	0	1	2	3
0	2 ⅓	2 ⅓	2 ⅓	--
1	--	0	--	3
BG	2	2	2	0
Agg	4 ⅓	4 ⅓	4 ⅓	3

Step 3: Balancing Port Group 0 (again)

Figure 5: Port group rebalancing algorithm. Egress ports that are not a member of the port group are indicated by '—'. The last row of the matrix represents the aggregate traffic (Agg) on the member ports (from port counters). The row indicating background traffic (BG) is added for clarity and is not directly measured by Dahu.

switch generates a *port group traffic matrix* where the rows represent port groups and columns represent egress ports in the switch (see Figure 5). The elements in a row represent egress ports and the amount of traffic (bandwidth) that the port group is currently forwarding to those egress ports. If an egress port is not a member of the port group corresponding to the matrix row, then the respective matrix element is zeroed. Additionally, the *Aggregate utilization* row of elements stores the total bandwidth utilization on each egress port. This is the bandwidth based on the egress port counter, and accounts for traffic forwarded by any of the port groups, as well as background traffic on the port that is not re-routable using port groups, such as elephant flows pinned to the path by Hedera.

We first pick a port group in the matrix and try to balance the aggregate traffic for each of the member ports by moving traffic from this port group to different member ports where possible. To do this, Dahu computes the average aggregate utilization of all member ports of the port group. Then, it reassigns the traffic for that port group to equalize the aggregate traffic for the member ports to the extent possible. If a member port's aggregate traffic exceeds the average across all members, and the member port receives some traffic from this port group then we reassign the traffic to other member ports as appropriate. Dahu performs this operation for all port groups and repeats until convergence. To ensure convergence, we terminate the algorithm when subsequent iterations offload less than a small constant threshold δ. Figure 5 shows the steps in the algorithm. Host facing ports in the switch can be ignored when executing this algorithm.

4.5 Load Balancing Heuristic

The load balancing algorithm considers all physical ports and port groups in the switch and aims to balance the aggregate load on all of them to the extent possible. However, the algorithm may take many steps to converge for a large switch with many ports and port groups. We now describe a quick and practical heuristic to compute the balanced traffic allocation. The key idea behind the heuristic is to offload traffic from the highest loaded port to the least loaded port with which it shares membership in any of the port group, instead of trying to balance the aggregate load on all ports. By running the heuristic quickly, the switch can balance the port groups at time scales on the order of a few milliseconds.

The heuristic, as described in Figure 6, is repeated for some fixed number R (say 16) of highest loaded switch ports, and has a low runtime of around 1ms. The runtime depends on the number of physical ports and port groups in the switch and is independent of the number of flows in the system. Our research grade implementation of the heuristic for our simulator running on a general purpose x86 CPU showed average runtimes of few 10's of microseconds to 0.5 milliseconds, even for large networks with over 130,000 servers. We believe an optimized version targeted at a switch ARM or PPC processor can run within 1ms with a small DRAM requirement of under 10 MB. In the rest of this paper, we employ this heuristic for load balancing.

1. Sort the physical ports by their aggregate utilization
2. Offload traffic from the highest loaded port H1 to the least loaded port with which it shares membership in any port group
3. Continue offloading traffic from H1 to the least loaded ports in order until they are completely balanced or H1 runs out of lesser loaded ports to offload to.

Figure 6: Load Balancing Heuristic

4.6 Fault Tolerance

Dahu relies on link-level techniques for fault detection, and uses existing protocols to propagate fault updates through the network. If a particular egress link or physical port P_f on the switch is down, the virtual ports in each port group which map to P_f are remapped to other member ports of the respective port groups. The remapping is performed by switch software and the actual policy could be as simple as redistributing the virtual ports uniformly to other egress ports or something more complicated.

On the other hand, when the switch receives fault notifications from the rest of the network, a specific egress port P_f may have to be disabled for only some destination prefixes because of downstream faults. We use the allowed port bitmaps technique described in Section 3.2 to just disable P_f for specific prefixes. The virtual port to physical port mappings in the port groups are left unchanged. In both scenarios, the only flows migrated to other egress ports are ones that were earlier mapped to the failed egress port P_f. When a physical port comes up, some virtual ports automatically get mapped to it the next time port groups are balanced.

5. DEPLOYABILITY

Deployability has been an important goal during the design of Dahu. In this section, we look at two primary requirements for

adding Dahu support to switches: the logic to implement the functionality, and the memory requirements of the data structures.

To our knowledge, existing switch chips do not provide Dahu-like explicit hardware support for non-minimal routing in conjunction with dynamic traffic engineering. However, there are some similar efforts including Broadcom's resilient hashing feature [9] in their modern switch chips which is targeted at handling link failure and live topology updates, and the Group Table feature in the recent OpenFlow 1.1 Specification [24] which uses a layer of indirection in the switch datapath for multipath support. The increasing popularity of OpenFlow, software defined networks [22], and custom computing in the control plane (via embedded ARM style processors in modern switch silicon) indicates a new trend that we can leverage where large data centers operators are adopting the idea of a programmable control plane for the switches. The need for switch hardware modification to support customizable control plane for switches is no longer a barrier to innovation, as indicated by the deployment of switches with custom hardware by companies like Google [18].

To implement the hardware logic, we also need sufficient memory in the chip to support the state requirements for Dahu functionality. We now briefly estimate this overhead. Consider a large Dahu switch with 128 physical ports, 64 port groups with 1,024 virtual ports each, 16,384 prefixes in the routing table, and support for up to two different allowed port bitmaps for each prefix. The extra state required for all of Dahu's features is a modest 640 KB. Of this, 64 KB each are required for storing the virtual to physical port mappings for all the port groups, and the port group counters per egress port. 512 KB is required for storing two bitmaps for each destination prefix. A smaller 64 port switch would only need a total of 352 KB for a similar number of port groups and virtual ports. This memory may come at the expense of additional packet buffers (typically around 10 MB); however, recent trends in data center congestion management [4, 5] indicate that trading a small amount of buffer memory for more adaptive routing may be worthwhile.

6. EVALUATION

We evaluated Dahu through flow-level simulations on both HyperX and Fat-tree topologies. Overall, our results show:

1. 10-50% throughput improvement in Fat-tree networks, and 250-500% improvement in HyperX networks compared to ECMP.
2. With an increase of only a single network hop, Dahu achieves significant improvements in throughput.
3. Dahu scales to large networks of over 130,000 nodes.
4. Dahu enables MPTCP to leverage non-shortest paths and achieve higher throughput with fewer subflows.

The evaluation seeks to provide an understanding of Dahu's effect on throughput and hop count in different network topologies (HyperX and Fat-tree) under different traffic patterns. We first present a description of the simulator that we used for our experiments and the methodology for validating its accuracy. We simulate HyperX networks, large and small, and measure throughput as well as expected hop count for different workloads. We then move on to evaluate Dahu on an 8,192 host Fat-tree network using two communication patterns. We conclude this section by evaluating how MPTCP benefits from Dahu through the use of non-shortest paths.

6.1 Simulator

We evaluated Dahu using a flow level network simulator that models the performance of TCP flows. We used the flow level simulator from Hedera [3], and added support for decentralized routing in each switch, port groups, allowed port bitmaps, and the load balancing heuristic. The Dahu-augmented Hedera simulator evaluates the AIMD behavior of TCP flow bandwidths to calculate the total throughput achieved by flows in the network.

We built a workload generator that generates open-loop input traffic profiles for the simulator. It creates traffic profiles with different distributions of flow inter-arrival times and flow sizes. This allows us to evaluate Dahu's performance over a wide range of traffic patterns, including those based on existing literature [6, 16]. Modeling the AIMD behavior of TCP flow bandwidth instead of per-packet behavior means that the simulator does not model TCP timeouts, retransmits, switch buffer occupancies and queuing delay in the network. The simulator only models unidirectional TCP data flows but not the reverse flow for ACKs. We believe this is justified, since the bandwidth consumed by ACKs is quite negligible compared to data. We chose to make these trade-offs in the simulator to evaluate at a large scale—over 130K servers, which would not have been possible otherwise.

We simulated five seconds of traffic in each experiment, and each switch rebalanced port groups (16 highest loaded ports) and recomputed prefix bitmaps every 10ms. For non-shortest path forwarding, switches used 80% of available capacity along shortest paths to the destination as the threshold utilization to dynamically enable non-shortest paths. These values were chosen based on empirical measurements.

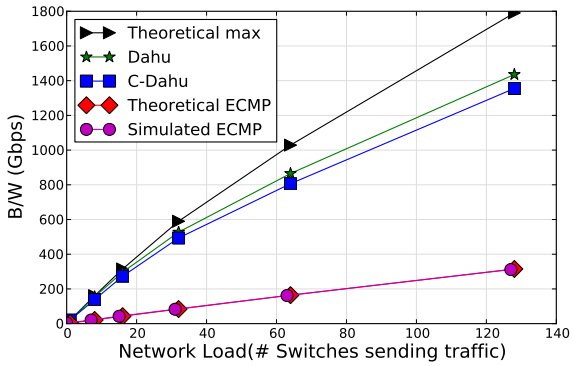


Figure 7: Simulator throughput vs. theoretical maximum

Simulator Validation: To validate the throughput numbers reported by the simulator, we generated a range of traffic profiles with a large number of long-lived flows between random hosts in a (L=3, S=8, T=48) HyperX network with 1Gbps links; (L, S, T defined in Section 4.1). We computed the theoretical maximum bandwidth achievable for the traffic patterns by formulating maximum multi-commodity network flow problems and solving them using the CPLEX [10] linear program solver—both for shortest path routing and non-minimal routing. We also ran our simulator on the same traffic profile.

As shown in Figure 7 the aggregate throughput reported by the simulator was within the theoretical maximum for all the traffic patterns that we validated. In case of shortest path forwarding, the theoretical and simulator numbers matched almost perfectly indicating that the ECMP implementation was valid. With non-minimal forwarding, the simulator's performance is reasonably close to the theoretical limit. Note that the multi-commodity flow problem simply optimizes for the total network utilization whereas the simulator and TCP in general, also take fairness into account.

In addition, we also explicitly computed the max-min fair flow bandwidths for these traffic profiles using the water-filling algorithm [8]. We compared the resulting aggregate throughput to those reported by the simulator. For all evaluated traffic patterns, the simulator throughput was within 10% of those reported by the max-min validator. This small difference is because the TCP's AIMD congestion control mechanism only yields approximate max-min fairness in flow bandwidths whereas the validator computes a perfectly max-min fair distribution.

6.2 HyperX Networks

We first evaluate Dahu with HyperX networks which have many paths of differing lengths between any source and destination. We simulate a (L=3, S=14, T=48) HyperX network with 1Gbps links, as described in [1]. This models a large data center with 131,712 servers, interconnected by 2,744 switches, and an oversubscription ratio of 1:8.

We seek to measure how Dahu's non-minimal routing and load balancing affect performance as we vary traffic patterns, the maximum derouting threshold, and non-minimal routing scheme (constrained or not). We run simulations with Clique and Mixed traffic patterns (described next), and compare the throughput, average hop count and link utilizations for Dahu and ECMP. In the graphs, Dahu-n refers to Dahu routing with at most n deroutings. C-Dahu-n refers to the similar Constrained Dahu routing variant.

6.2.1 Clique Traffic Pattern

A *Clique* is a subset of switches and associated hosts that communicate among themselves; each host communicates with every other host in its clique over time. This represents distributed jobs in a data center which are usually run on a subset of the server pool. A typical job runs on a few racks of servers. There could be multiple cliques (or jobs) running in different parts of the network. We parameterize this traffic pattern by i) clique size, the number of switches in the clique, and ii) total number of cliques in the network. In this experiment, we vary the total number of cliques from 64 to 768, keeping the clique size fixed at 2 switches (96 servers). Each source switch in a clique generates 18Gbps of traffic with 1.5 MB average flow size.

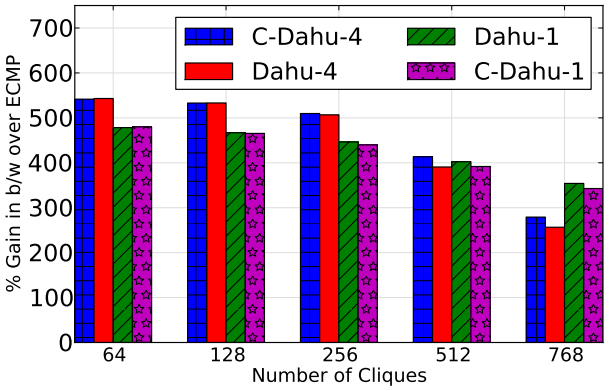
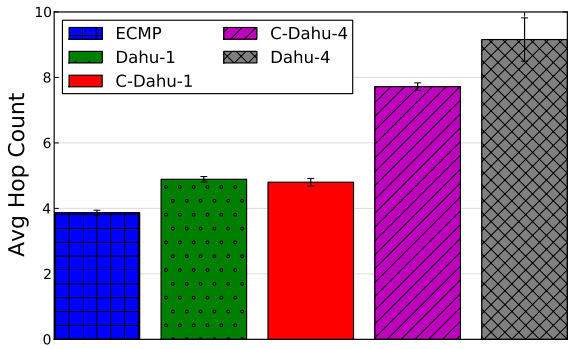
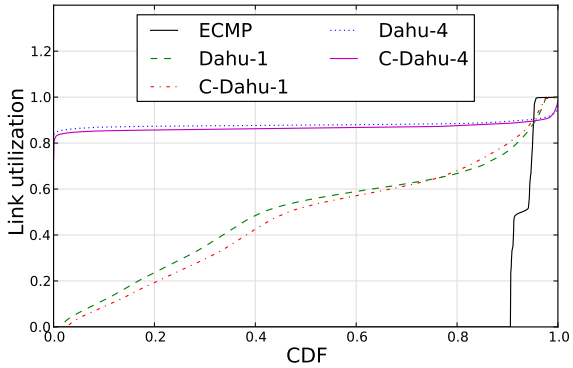


Figure 8: Throughput gain with Clique traffic pattern.

Bandwidth: Figure 8 shows the bandwidth gains with Dahu relative to ECMP as we vary the number of communicating cliques. Dahu offers substantial gains of 400-500% over ECMP. The performance gain is highest with a smaller number of cliques, showing that indeed derouting and non-shortest path forwarding can effectively take advantage of excess bandwidth in HyperX networks.



(a) Average hop count



(b) Link utilization (number of cliques = 512)

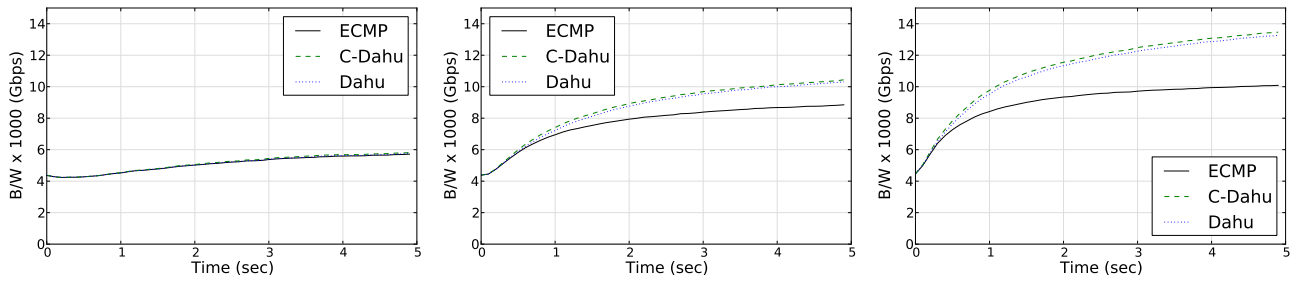
Figure 9: Average hop count, and link utilization for Clique traffic pattern with different routing schemes

This validates a major goal of this work, which is improving the statistical multiplexing of bandwidth in direct network topologies. As the number of cliques increases, the bandwidth slack in the network decreases, and the relative benefit of non-minimal routing comes down to around 250%. Dahu and constrained Dahu have similar performance for the same derouting threshold.

We further find that a large derouting threshold provides larger benefit with less load, since there are many unused links in the network. As load increases, links are more utilized on average, bandwidth slack reduces, and a derouting threshold of one starts performing better.

Hop count: Beyond raw throughput, latency in an important performance metric that is related to network hop count. Figure 9a shows the average hop count for each routing scheme. Dahu delivers significantly higher bandwidth with a small increase in average hop count. Average hop count increases with increase in derouting threshold. For smaller derouting threshold, the hop count is similar to that of ECMP while still achieving most of the bandwidth improvements of non-minimal routing. Note that the small error bars indicate that the average hop count is similar while varying the number of cliques.

Link utilization: Figure 9b shows the CDF of inter-switch link utilizations for ECMP and Dahu for the experiment with 512 cliques. With shortest path routing, 90% of the links have zero utilization, whereas Dahu achieves its bandwidth gains by utilizing available capacity on additional links in the network. Also, we see that a single derouting can achieve most of the overall bandwidth gains while consuming bandwidth on significantly fewer links in the network thereby sparing network capacity for more traffic.



(a) Load = 4.5Gbps/switch **(b)** Load = 17.5Gbps/switch **(c)** Load = 33.5Gbps/switch

Figure 10: Dahu with Mixed Traffic Pattern

6.2.2 *Mixed Traffic Pattern*

The "mixed traffic pattern" represents an environment with a few *hot racks* that send lot of traffic, representing jobs like data backup. For this traffic pattern, we simulate 50 cliques with 10 switches in each. Every switch acts as a network hot-spot and has flows to other members of the clique with average size = 100 MB. The load from each source switch is varied from 3Gbps to 32Gbps. We also generate random all-to-all traffic between all the hosts in the network. This background traffic creates an additional load of 1.5Gbps per source switch with average flow size of 200 KB.

Figure 10 shows that for low load levels (4.5Gbps total load per switch) ECMP paths are sufficient to fulfill demand. As expected, total bandwidth achieved is same for both ECMP and Dahu. However, at high load (17.5Gbps and 33.5Gbps per switch) Dahu performs significantly better than ECMP by utilizing available slack bandwidth.

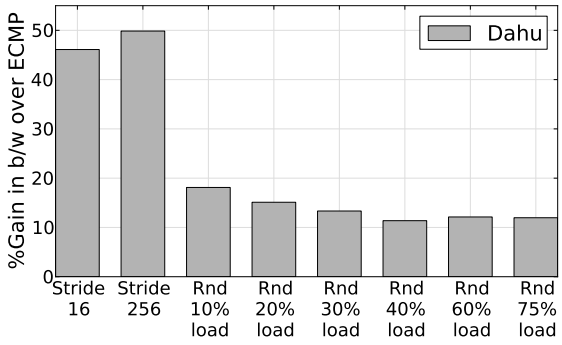


Figure 11: Throughput gain for k = 32 Fat-tree with Stride and Random (Rnd) traffic patterns.

6.3 Fat-Tree Networks

To illustrate Dahu's generality, we evaluate it in the context of a Fat-tree topology. Fat-trees, unlike HyperX, have a large number of shortest paths between a source and destination, so this evaluation focuses on Dahu's load balancing behavior, rather than its use of non-shortest paths. We compare Dahu with ECMP, with hosts communicating over long lived flows in a k = 32 Fat-tree (8,192 hosts). We consider these traffic patterns: (1) *Stride*: With n total hosts and stride length x, each host i sends traffic to host $(i + x) \bmod n$. (2) *Random*: Each host communicates with another randomly chosen destination host in the network. To study the effect of varying overall network load, we pick a subset of edge switches that send traffic to others and vary the number of hosts on each of these edge switches that originate traffic.

Figure 11 shows that Dahu achieves close to 50% improvement with stride traffic patterns. The load balancing heuristic rebalances virtual port mappings at each switch minimizing local hash imbalances and improves total throughput. For random traffic patterns, Dahu outperforms ECMP by 10-20%. Overall, Dahu is better able to utilize network links in Fat-tree networks than ECMP, even when only shortest-path links are used.

6.4 MPTCP in HyperX Networks

MPTCP is a recent host-based transport layer solution for traffic engineering [26]. MPTCP relies on splitting each flow into multiple subflows that take different paths through the network, and modulates the amount of data transmitted on each subflow based on congestion, thus improving the network utilization. In this section, we evaluate how MPTCP benefits from Dahu non-shortest path routing, and the additional improvements achieved using Dahu dynamic load balancing.

We extended the *htsim* packet level simulator [11] (used to evaluate MPTCP in [26]), to simulate a (L=3, S=10, T=20) HyperX network with 100 Mbps links. This network has 1000 switches, 20,000 hosts and an oversubscription ratio of 1:4. We chose a smaller topology and lower link speed due to the higher computational overhead of packet level simulations. We generated a random permutation matrix (without replacement), and selected a subset of source-destination pairs to create long lived flows, with 50% of total hosts sending traffic. To evaluate the impact of non-shortest path routing, we ran MPTCP under two scenarios: (1) ECMP style shortest-path routing (MPTCP-ECMP), and (2) Dahu-style non-shortest path routing with one allowed deroute but no load balancing (MPTCP-1Der). To understand the additional impact of Dahu's load balancing, we also ran the Dahu simulator on the same topology and traffic pattern by treating each subflow as an independent TCP flow.

Since we used a packet level simulator for MPTCP and a flow level simulator for Dahu, we also validated that the two simulators reported comparable throughput results under identical scenarios [25, pg.12].

Figure 12 shows that Dahu's non-shortest path routing unlocks 300% more bandwidth compared to ECMP. With MPTCP-1Der, throughput increases with the number of subflows, indicating that in order to effectively leverage the large path diversity in direct connect networks, MPTCP needs to generate a large number of subflows, making it unsuitable for short flows. Dahu, on the other hand, is able to achieve a similar throughput with 8 subflows that

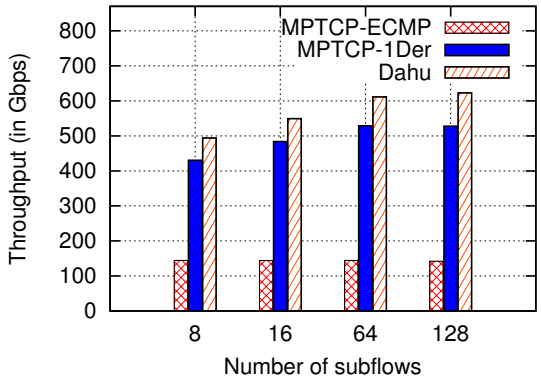


Figure 12: MPTCP-ECMP, MPTCP-1Der, and Dahu performance for L=3, S=10, T=20 HyperX topology. Results obtained from packet level simulations for MPTCP and flow level simulations for Dahu.

MPTCP-1Der achieves with 64 or 128 subflows, and can also handle short flows with efficient hash rebalancing. At the transport layer, MPTCP has no way of distinguishing between shortest and non-shortest paths and can leverage Dahu for better route selection. These results indicate that Dahu effectively enables MPTCP to leverage non-shortest paths, and achieve much better network utilization in direct networks with fewer subflows.

7. DISCUSSION

As seen in Section 6, Dahu exploits non-minimal routing to derive large benefits over ECMP for different topologies and varying communication patterns. Yet, there is a scenario where non-minimal routing can be detrimental. This occurs when the network as a whole is highly saturated; shortest path forwarding itself does well as most links have sufficient traffic and there is no "unused" capacity or slack in the network. With Dahu, a derouted flow consumes bandwidth on more links than if it had used just shortest paths, thereby contributing to congestion on more links. In large data centers, this network saturation scenario is uncommon. Networks have much lower average utilizations although there may be hot-spots or small cliques of racks with lot of communication between them. Usually, there is network slack or unused capacity which Dahu can leverage. The network saturation case can be dealt with in many ways. For example, a centralized monitoring infrastructure can periodically check if a large fraction of the network is in its saturation regime and notify switches to stop using non-minimal paths.

Alternatively, a simple refinement to the localized load balancing scheme can be used which relies on congestion feedback from neighboring switches to fall back to shortest path forwarding in such high load scenarios. Network packets are modified to store 1 bit in the IP header which is updated by each switch along the path of a packet to indicate whether the switch used a shortest path egress port or derouted the packet. A switch receiving a packet checks if two conditions are satisfied: (1) It doesn't have enough capacity to the destination along shortest paths alone, and (2) The previous hop derouted the packet. If both conditions are satisfied, it sends congestion feedback to the previous hop notifying it to stop sending derouted traffic through this path for the particular destination prefix, for a certain duration of time (say 5ms). This solution is discussed further in [25].

8. RELATED WORK

There have been many recent proposals for scale-out multipath data center topologies such as Clos networks [2, 16, 23], direct networks like HyperX [1], Flattened Butterfly [20], DragonFly [21], and even randomly connected topologies proposed in Jellyfish [27]. Many current proposals use ECMP-based techniques which are inadequate to utilize all paths, or to dynamically load balance traffic. Routing proposals for these networks are limited to shortest path routing (or K-shortest path routing with Jellyfish) and end up underutilizing the network, more so in the presence of failures. While DAL routing [1] allows deroutes, it is limited to HyperX topologies. In contrast, Dahu proposes a topology-independent, deployable solution for non-minimal routing that eliminates routing loops, routes around failures, and achieves high network utilization.

Hedera [3] and MicroTE [7] propose a centralized controller to schedule long lived flows on globally optimal paths. However they operate on longer time scales and scaling them to large networks with many flows is challenging. While DevoFlow [12] improves the scalability through switch hardware changes, it does not support non-minimal routing or dynamic hashing. Dahu can co-exist with such techniques to better handle congestion at finer time scales.

MPTCP [26] proposes a host based approach for multipath load balancing, by splitting a flow into multiple subflows and modulating how much data is sent over different subflows based on congestion. However, as a transport protocol, it does not have control over the network paths taken by subflows. Dahu exposes the path diversity to MPTCP and enables MPTCP to efficiently utilize the non-shortest paths in a direct connect network. There have also been proposals that employ variants of switch-local per-packet traffic splitting [13, 30]. With Dahu, instead of per-packet splitting, we locally rebalance flow aggregates across different paths thereby largely reducing in-network packet reordering.

Traffic engineering has been well studied in the context of wide area networks. TeXCP [19], MATE [14], and REPLEX [15] split flows on different paths based on load, however their long control loops make them inapplicable in the data center context which requires faster response times to deal with short flows and dynamic traffic changes. FLARE [28] exploits the inherent burstiness in TCP flows to schedule "flowlets" (bursts of packets) on different paths to reduce extensive packet reordering.

Finally, a key distinction between Dahu and the related traffic engineering approaches is that Dahu actively routes over non-shortest paths in order to satisfy traffic demand. Dahu decouples non-minimal routing and its mechanism for more balanced hashing and offers a more flexible architecture for better network utilization in direct connect networks.

9. CONCLUSION

Existing solutions for leveraging multipaths in the data center rely on ECMP which is insufficient due to its static nature and inability to extend beyond shortest path routing. We present a new switch mechanism, Dahu, that enables dynamic hashing of traffic onto different network paths. Dahu exploits non-shortest path forwarding to reduce congestion while preventing persistent forwarding loops using novel switch hardware primitives and control software. We present a decentralized load balancing heuristic that makes quick, local decisions to mitigate congestion, and show the feasibility of proposed switch hardware modifications. We evaluate Dahu using a simulator for different topologies and different traffic patterns and show that it significantly outperforms shortest path routing and complements MPTCP performance by selecting good paths for hashing subflows.

10. REFERENCES

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proc. of SC*, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. of Usenix NSDI*, 2010.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of ACM SIGCOMM*, 2010.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proc. of Usenix NSDI*, 2012.

[6] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proc. of ACM CoNEXT*, 2011.

[8] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1987.

[9] Broadcom Smart-Hash Technology. `http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf`.

[10] CPLEX Linear Program Solver. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[11] MPTCP htsim simulator. `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html`.

[12] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. of ACM SIGCOMM*, 2011.

[13] A. A. Dixit, P. Prakash, R. R. Kompella, and C. Hu. On the Efficacy of Fine-Grained Traffic Splitting Protocols in Data Center Networks. Technical Report Purdue/CSD-TR 11-011, 2011.

[14] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *Proc. of IEEE INFOCOM*, 2001.

[15] S. Fischer, N. Kammenhuber, and A. Feldmann. REPLEX: Dynamic Traffic Engineering Based on Wardrop Routing Policies. In *Proc. of ACM CoNEXT*, 2006.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable And Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, 2009.

[17] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. of ACM SIGCOMM*, 2010.

[18] U. Hölzle. OpenFlow @ Google. Talk at Open Networking Summit, 2012.

[19] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *Proc. of ACM SIGCOMM*, 2005.

[20] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A Cost-efficient Topology for High-radix networks. In *Proc. of ISCA*, 2007.

[21] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proc. of ISCA*, 2008.

[22] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform For Large-scale Production Networks. In *Proc. of Usenix OSDI*, 2010.

[23] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A Fault-Tolerant Engineered Network. In *Proc. of Usenix NSDI*, 2013.

[24] OpenFlow Switch Specification - Version 1.1. `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`.

[25] S. Radhakrishnan, R. Kapoor, M. Tewari, G. Porter, and A. Vahdat. Dahu: Improved Data Center Multipath Forwarding. Technical Report UCSD/CS2013-0992, Feb 2013.

[26] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proc. of ACM SIGCOMM*, 2011.

[27] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[28] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[29] Titan Supercomputer. `http://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/`.

[30] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. of ACM SIGCOMM*, 2012.